



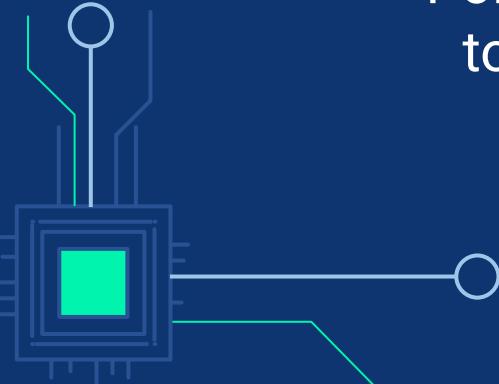
WORKSHOP DE REDES NEURAIS

Com PyTorch

01

POR QUE REDES NEURAIS?

Por que esses modelos computacionais se tornaram tão importantes em tão pouco tempo?



**“Scale has been driving Deep
Learning progress”**

ANDREW NG

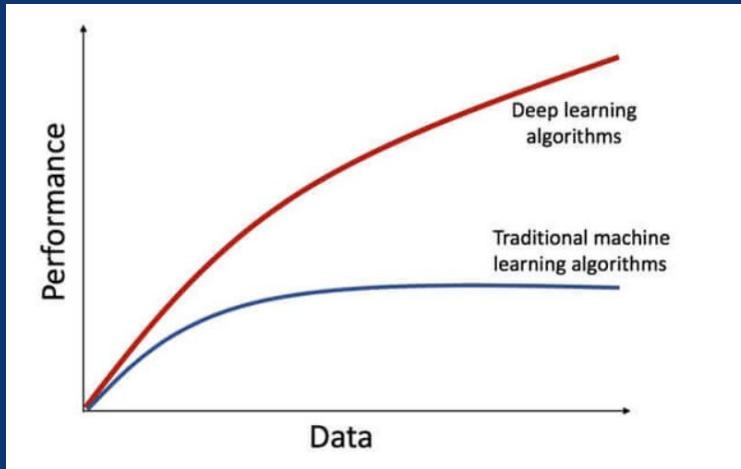




3 FATORES PRINCIPAIS

- 1. Diferença de Performance**
- 2. Quantidade de Dados**
- 3. Poder Computacional**

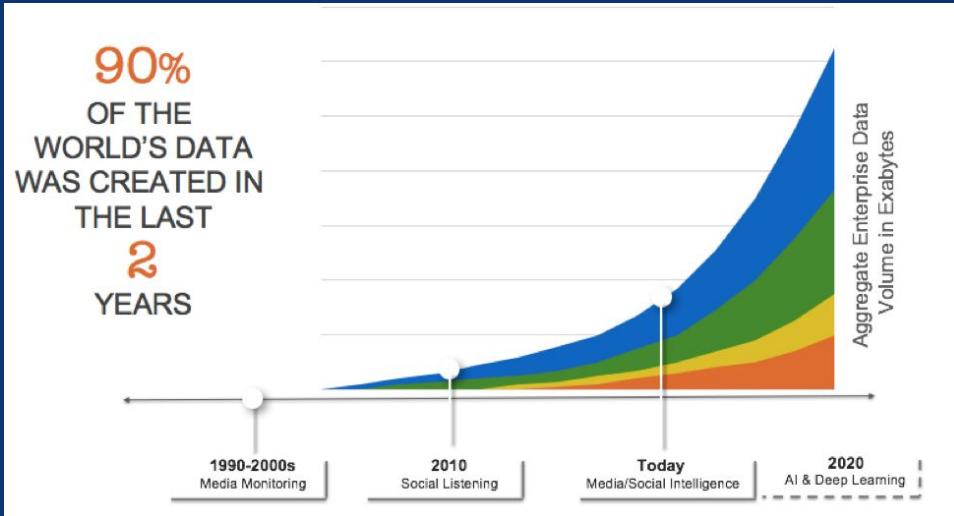
DIFERENÇA DE PERFORMANCE



Os modelos mais básicos de Machine Learning estabilizam a sua performance

Usando Redes Neurais, essa performance continua a aumentar com uma maior quantidade de dados, ou seja, **quanto mais dados melhor!**

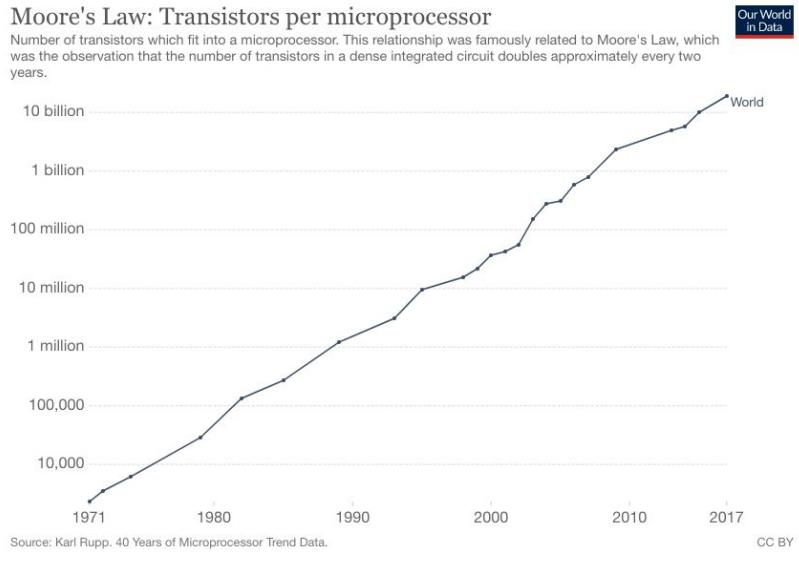
QUANTIDADE DE DADOS



Hoje, grande parte das atividades humanas foram digitalizadas, o que gera um grande aumento na produção de dados.

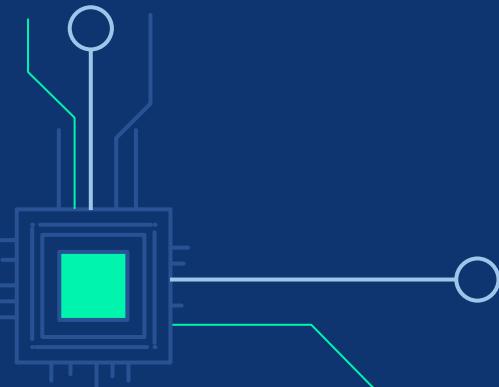
Com mais dados disponíveis, as redes neurais tornam-se a melhor opção para a resolução de problemas de grande escala.

PODER COMPUTACIONAL



Com o aumento do número de transistores em microprocessadores, o número de operações realizadas pelas máquinas cresceram enormemente, aumentando o poder computacional bruto e permitindo que redes neurais sejam treinadas em tempo hábil.

ESTRUTURA DA REDE



CLASSIFICAÇÃO BINÁRIA



CLASSIFICAÇÃO BINÁRIA

- Nós queremos saber se alguma coisa sobre nossa entrada é verdade ou não
- Ex: queremos saber se uma imagem é de um chihuahua ou não
- A saída da nossa função tem que dizer qual a probabilidade desse algo ser verdade ou não
- Sendo uma probabilidade, nossa saída deve ser um número entre 0 e 1



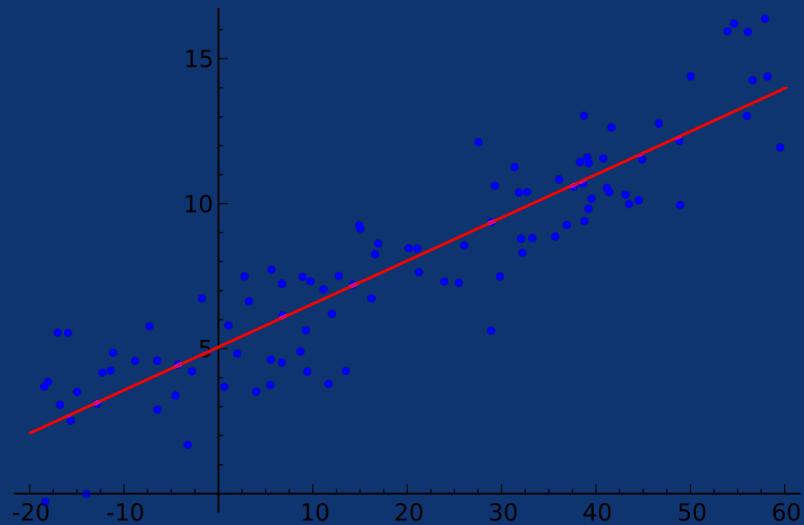
A imagem da esquerda deve dar uma probabilidade alta como 0.9 (90%), já a da direita, uma probabilidade baixa mais próxima de zero



REGRESSÃO LINEAR

- A regressão linear é a função mais simples que podemos testar
- Sozinha ela não satisfaz nossos requisitos para a classificação binária
- A saída (Y) não está contida entre 0 e 1

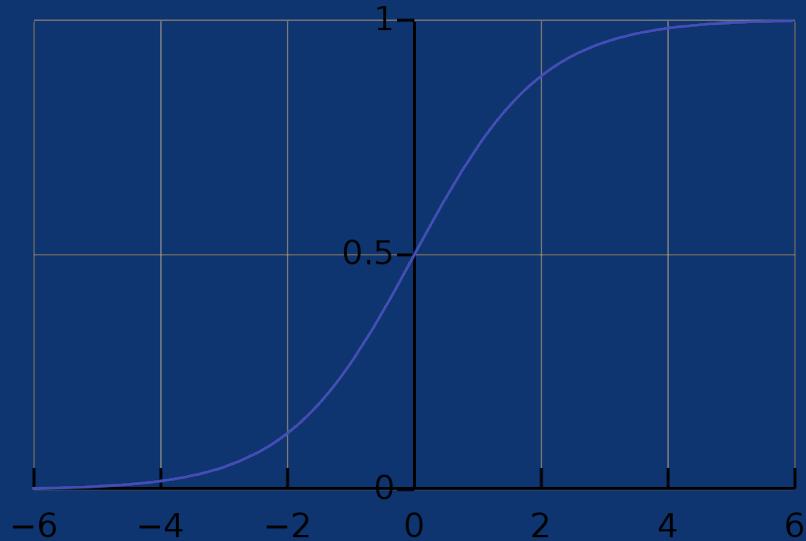
$$Y = a + bX$$



FUNÇÕES DE ATIVAÇÃO

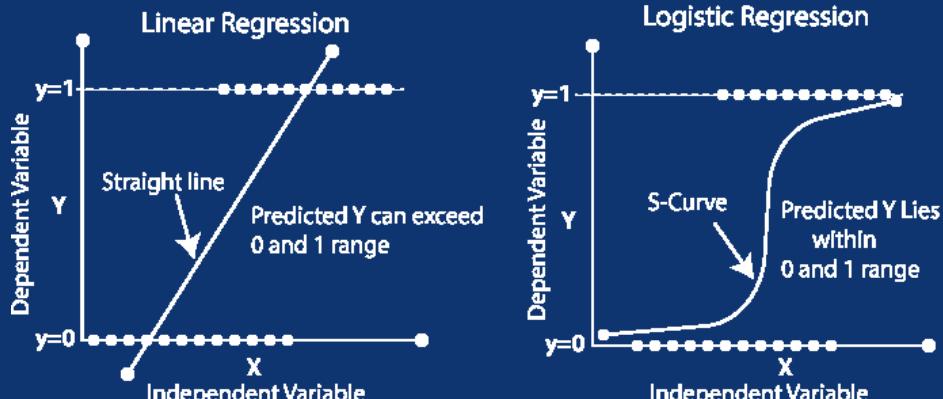
- Para resolver o problema da regressão linear, podemos passar sua saída por uma função de ativação
- Existem várias funções que podemos usar, mas nesse caso vamos usar a sigmoid (ou função logística)
- Fazendo a composição da regressão linear com função logística, obtemos as propriedades de que precisamos

$$f(x) = \frac{e^x}{1+e^x}$$



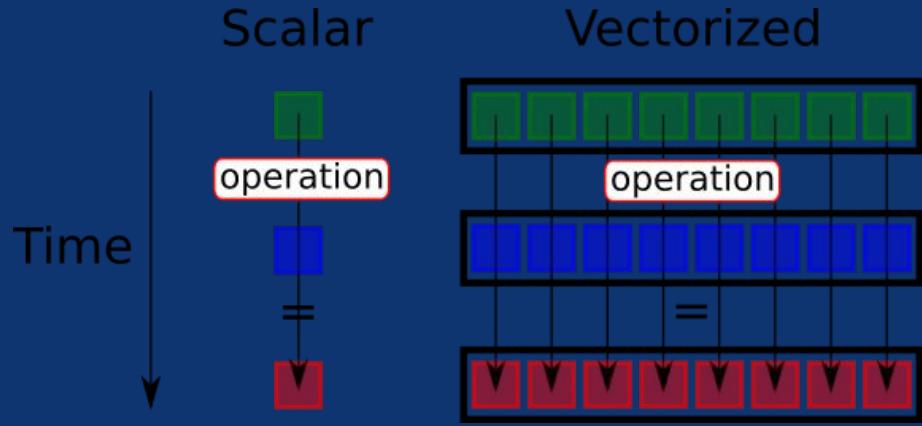
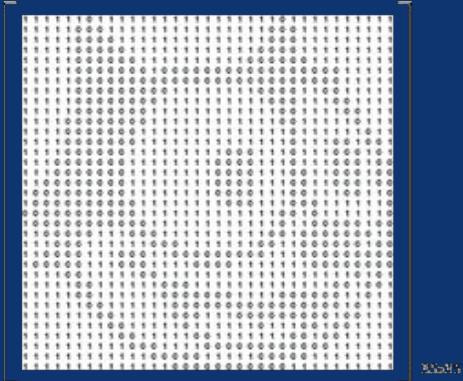
REGRESSÃO LOGÍSTICA

- Assim, chamamos essa composição de regressão logística
- Definindo um limiar (threshold), como 0,5 (50%), podemos fazer a classificação binária
- Ex.: Valores acima de 50% significam que a imagem é de um chiuaua, e não de qualquer outra coisa



VECTORIZATION

- Agora precisamos entender como trabalhar com vetores
- Nossas operações serão feitas com “vários números ao mesmo tempo”
- Isso permite que, por exemplo, passemos imagens como entradas de funções
- Para isso, basta transformar a matriz que representa a imagem em uma sequência de valores (vetor)



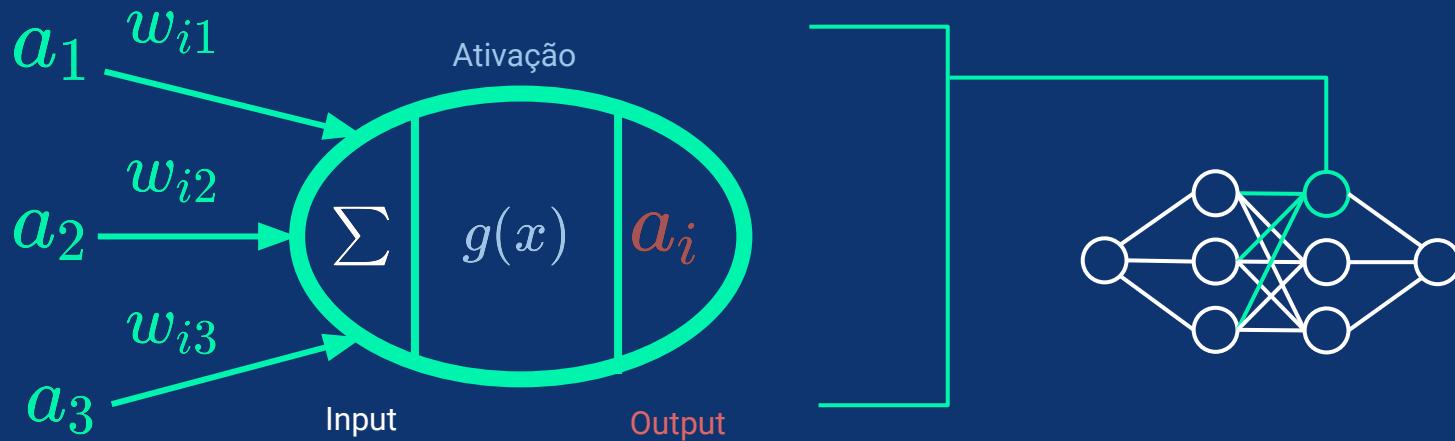
SOFTMAX

- Para podermos estender a funcionalidade do nosso classificador (saber se a imagem é de uma chiuaua, um bolinho ou uma preguiça, por exemplo), precisamos mudar nossa regressão
- No lugar da função logística, vamos usar uma função chamada Softmax.
- Essa função aceita um vetor de números como entrada, e nos devolve um valor que vamos chamar de ativação do neurônio
- Assim, normalizamos nossas saídas para o intervalo $[0, 1)$ e garantimos que a soma de todas elas é 1

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$



NEURÔNIO



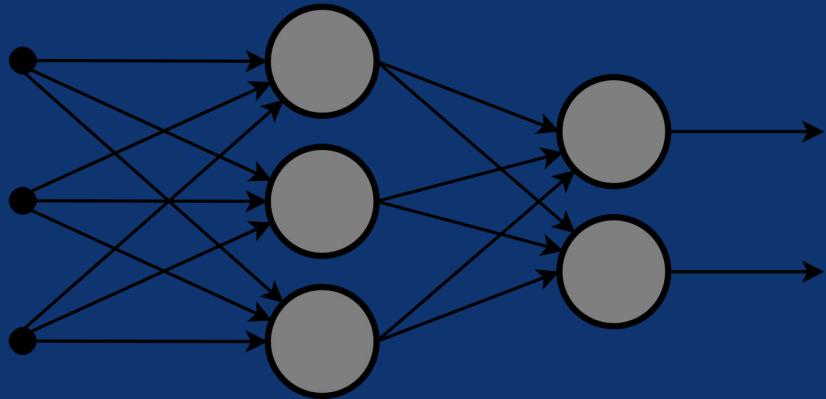
- modelo matemático que calcula uma soma ponderada das entradas, aplica uma função nessa soma e passa esse o resultado adiante

$$a_i = g(a_1 w_{i1} + a_2 w_{i2} + a_3 w_{i3} + b)$$



MONTANDO A REDE

- Juntando vários neurônios, temos nossa rede!
- Montamos ela de forma que cada neurônio passa seu valor de ativação para todos os neurônios da próxima camada



FORWARD PROPAGATION

É uma série de multiplicações de matrizes (transformações lineares) seguidas de funções de ativação para adicionar não-linearidade.

$$\phi \left(\begin{bmatrix} x_{11} & \dots & x_{1d} \\ x_{21} & \dots & x_{2d} \\ \vdots & \vdots & \vdots \\ x_{n1} & \dots & x_{nd} \end{bmatrix} \times \begin{bmatrix} w_{11} & \dots & w_{1m} \\ w_{21} & \dots & w_{2m} \\ \vdots & \vdots & \vdots \\ w_{d1} & \dots & w_{dm} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \right) = \begin{bmatrix} z_{11} \\ z_{12} \\ \vdots \\ z_{1m} \end{bmatrix}$$





GRADIENT DESCENT E BACKPROPAGATION

Como as redes aprendem

FUNÇÃO DE CUSTO

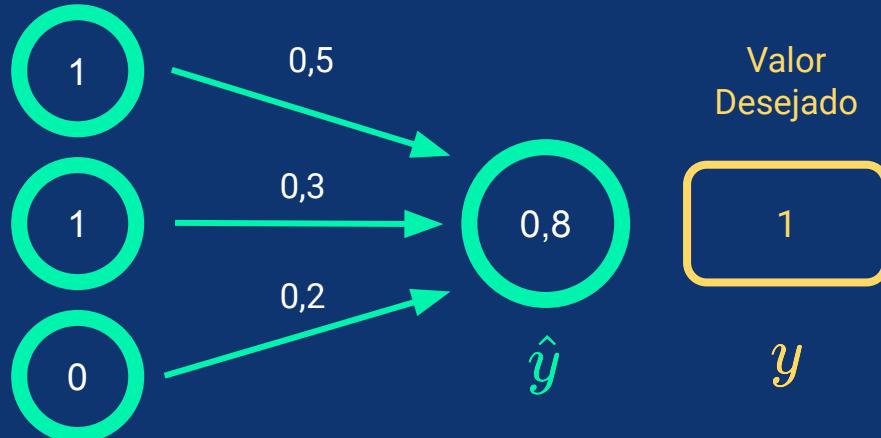
- No início do treinamento, os pesos da rede neural são **inicializados aleatoriamente**, gerando saídas basicamente aleatórias.
- Para ensinar nossa rede, é necessário analisar e corrigir essas previsões.
- Como avaliar a nossa rede neural?
- Estabelecemos uma **Função de custo**.



FUNÇÃO DE CUSTO - EXEMPLO

- Vamos utilizar a Função de Custo do **Erro Quadrático Médio**.
- O Erro Quadrático é o quadrado da diferença entre o valor predito e o valor desejado.

$$\text{Cost} = (\hat{y} - y)^2$$



$$\text{Cost} = (0,8 - 1)^2$$

$$\text{Cost} = (0,2)^2$$

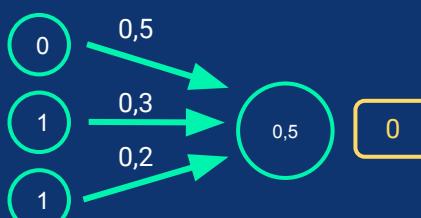
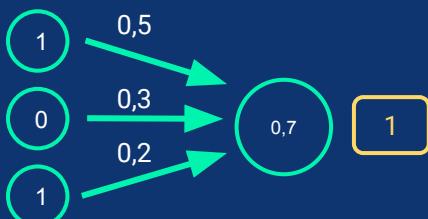
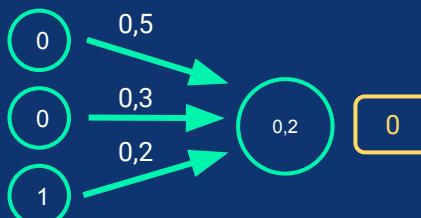
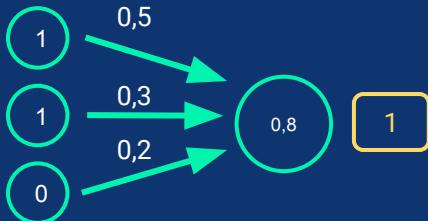
$$\text{Cost} = 0,04$$



FUNÇÃO DE CUSTO - EXEMPLO

- Entretanto, é importante que a rede neural tenha um baixo custo não em somente um caso, mas sim em todo o conjunto de dados.
- Dessa forma, a função de custo será a média dos erros quadráticos de todos os exemplos.

$$\text{Cost} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$



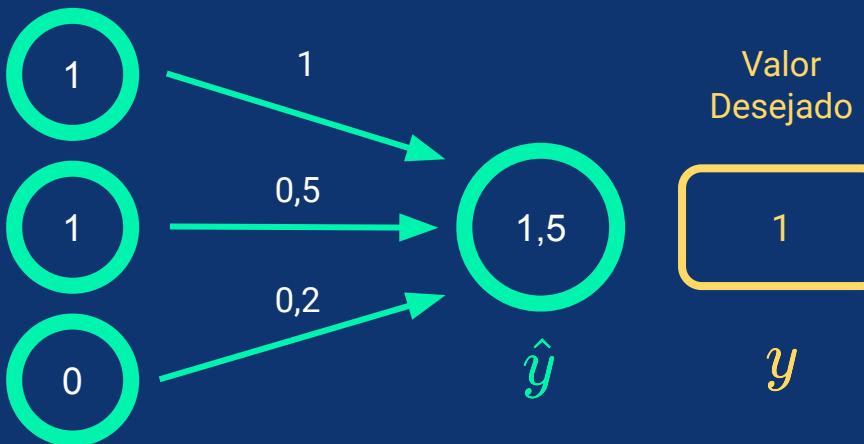
$$\text{Cost} = \frac{0,04+0,09+0,04+0,25}{4}$$

$$\text{Cost} = 0,105$$



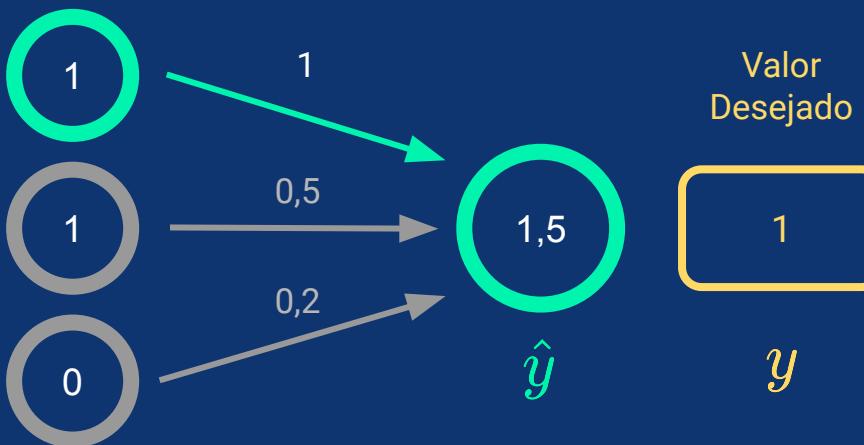
GRADIENT DESCENT

- Como otimizar os pesos da nossa rede neural para minimizar a função de custo?
- Vamos analisar somente um peso:



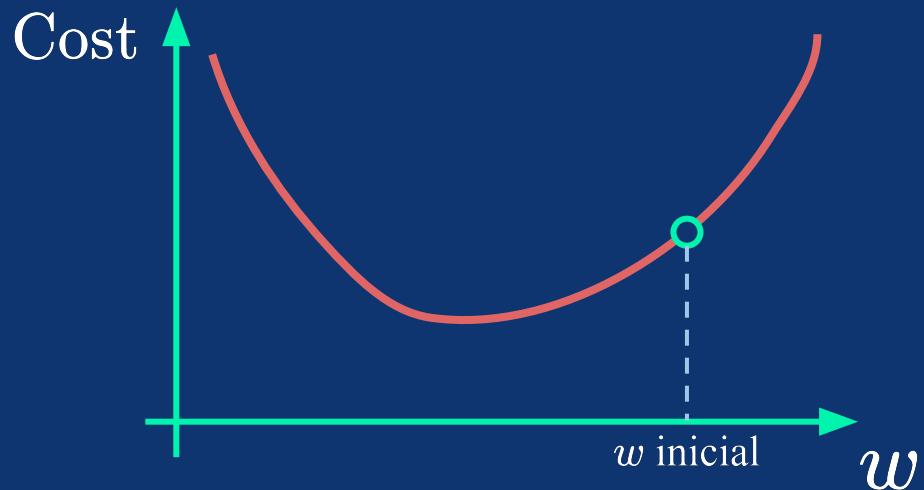
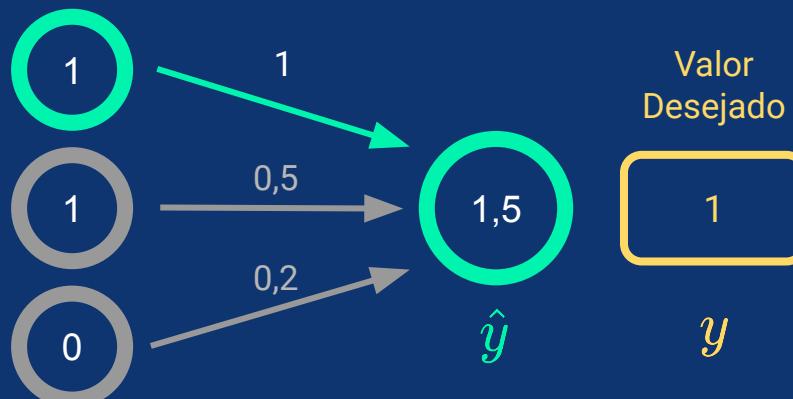
GRADIENT DESCENT

- Como otimizar os pesos da nossa rede neural para minimizar a função de custo?
- Vamos analisar somente um peso:



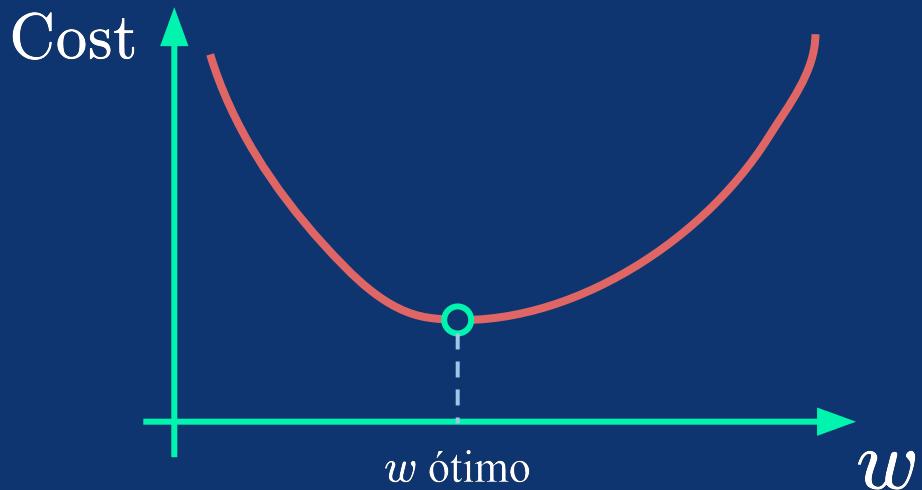
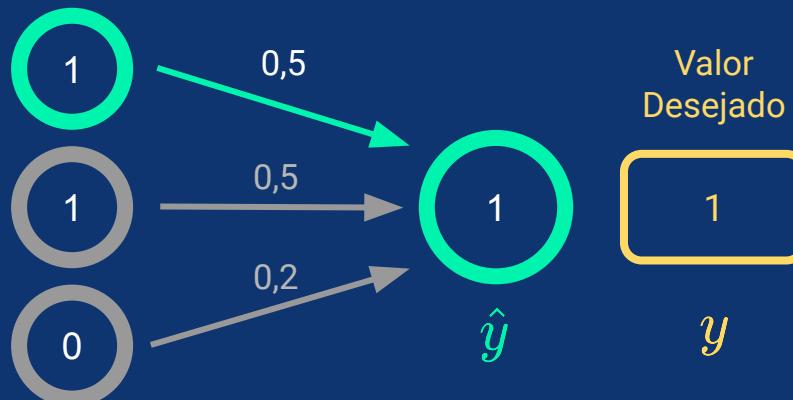
GRADIENT DESCENT

- Gráfico Função de Custo para um peso:



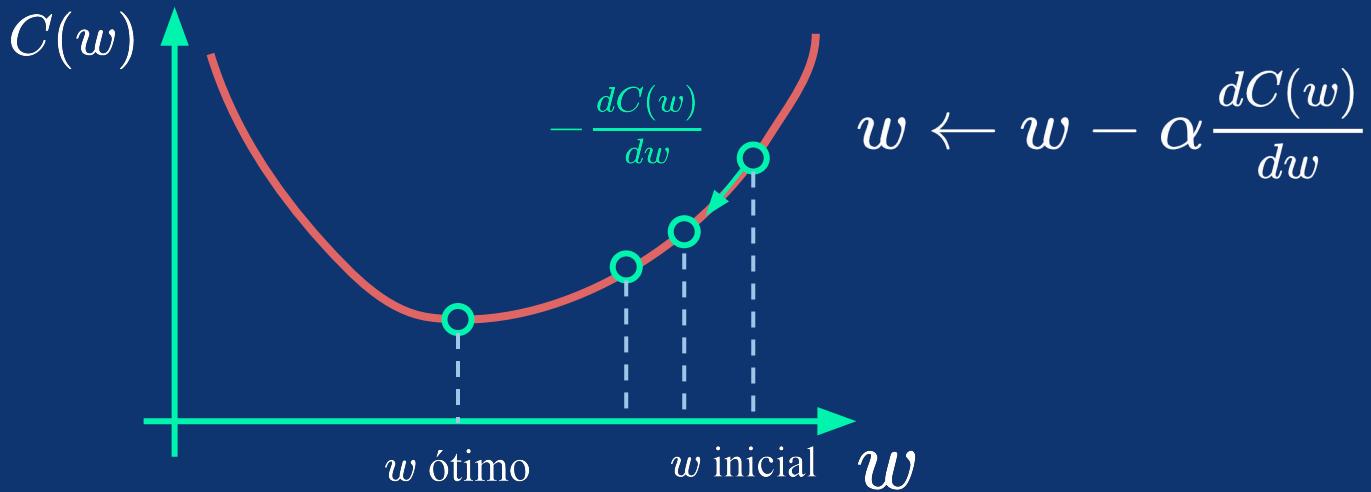
GRADIENT DESCENT

- Gráfico Função de Custo para um peso:



GRADIENT DESCENT

- Gradiente Descendente:
 - Descer a curva do gráfico da Função de Custo até atingir um mínimo.
 - Para isto, utilizamos as **derivadas** da função.



GRADIENT DESCENT

- Exemplo numérico:



$$C(w) = (\hat{y} - y)^2$$

$$\frac{dC(w)}{dw} = 2w - 1$$

$$w \leftarrow w - \alpha(2 \cdot w - 1)$$

$$w \leftarrow 1 - 0.2(2 - 1)$$

$$w \leftarrow 0.8$$



GRADIENT DESCENT

- Exemplo numérico:



$$C(w) = (\hat{y} - y)^2$$

$$\frac{dC(w)}{dw} = 2w - 1$$

$$w \leftarrow w - \alpha(2 \cdot w - 1)$$

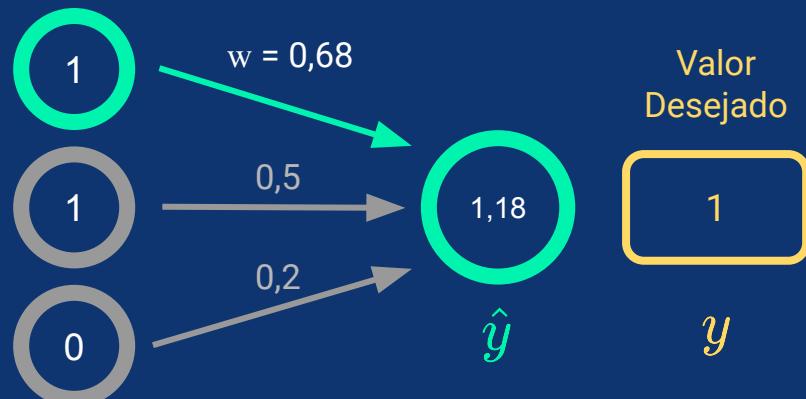
$$w \leftarrow 0.8 - 0.2(1.6 - 1)$$

$$w \leftarrow 0.68$$



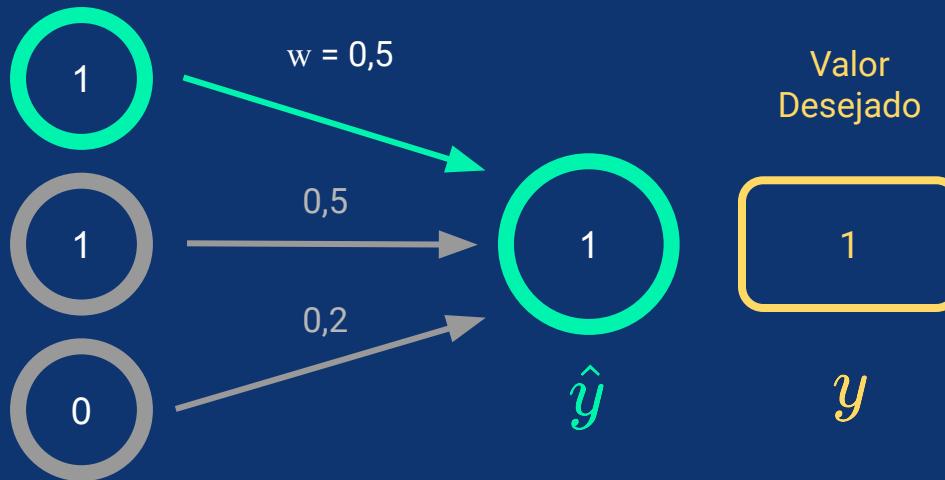
GRADIENT DESCENT

- Exemplo numérico:



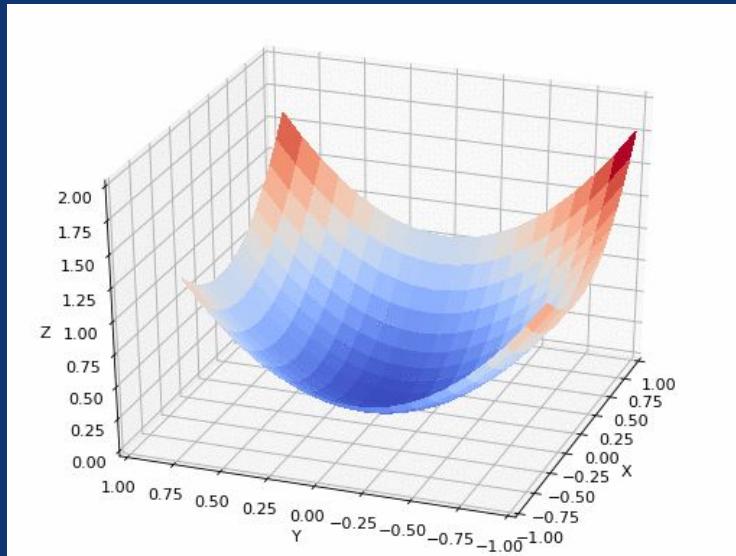
GRADIENT DESCENT

- Após inúmeras descidas de gradiente:



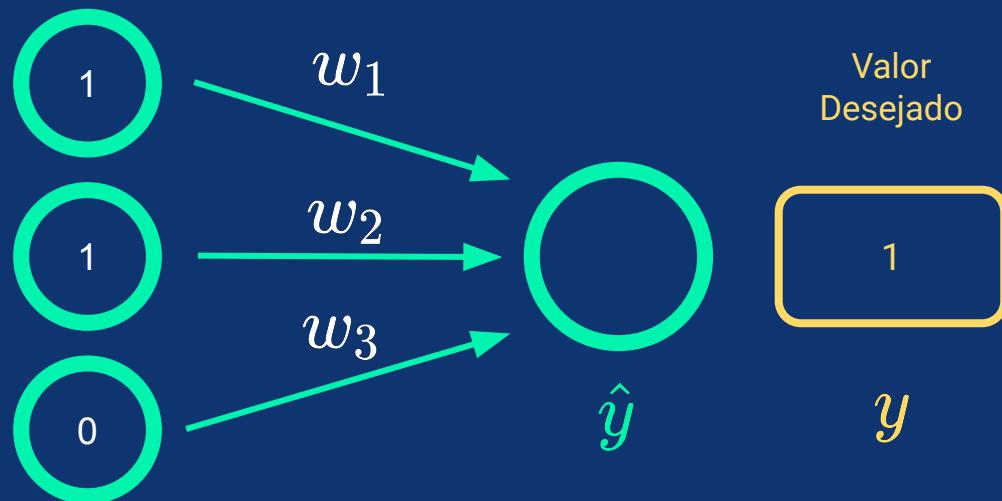
GRADIENT DESCENT

- Queremos otimizar todos os pesos da rede neural ao mesmo tempo.
- Isso significa que a nossa função de custo terá mais dimensões.



GRADIENT DESCENT

- Como funcionam os cálculos para mais dimensões?
- Representamos os nossos pesos como um **vetor**.



$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$



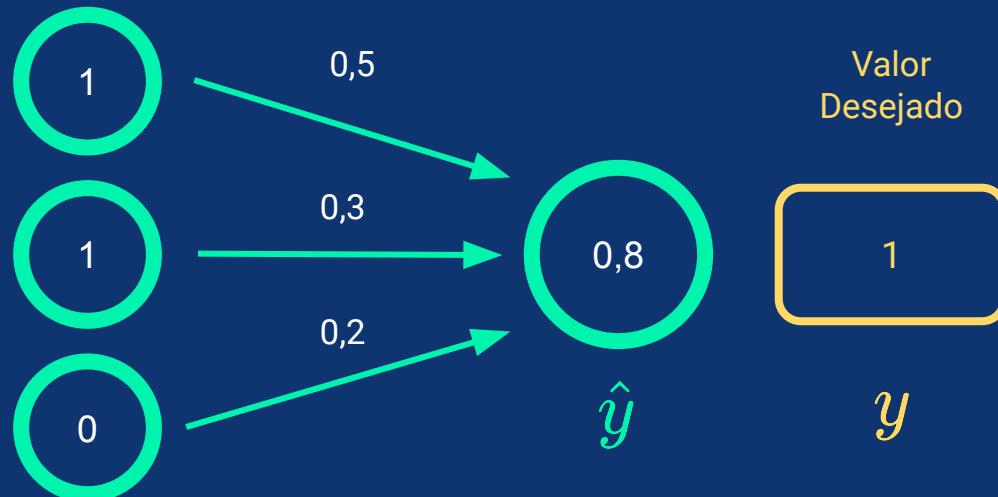
GRADIENT DESCENT

- Usaremos agora o **gradiente** da nossa função com relação aos pesos.



GRADIENT DESCENT

- Exemplo numérico:



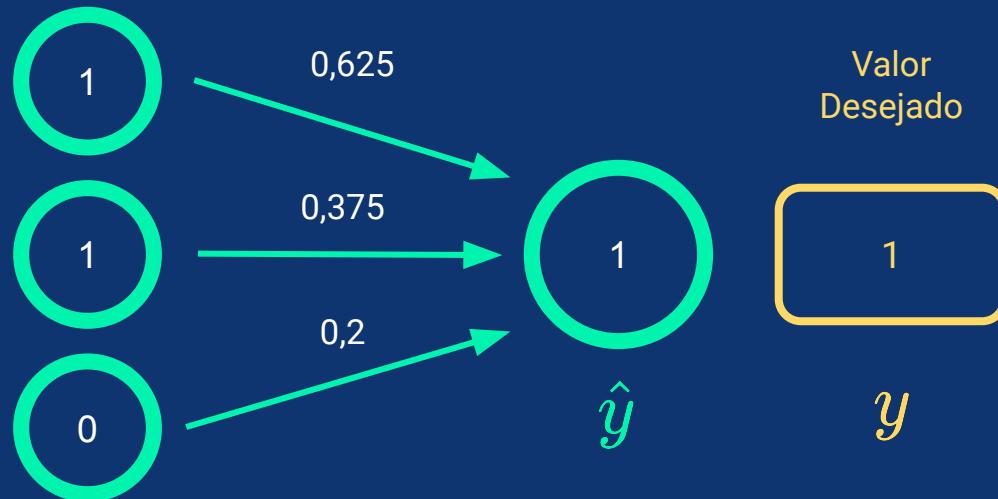
$$\vec{w} \leftarrow \vec{w} - \alpha \nabla C(\vec{w})$$

$$\begin{bmatrix} 0,625 \\ 0,375 \\ 0,2 \end{bmatrix} \leftarrow \begin{bmatrix} 0,5 \\ 0,3 \\ 0,2 \end{bmatrix} + \begin{bmatrix} 0,125 \\ 0,075 \\ 0 \end{bmatrix}$$



GRADIENT DESCENT

- Após a atualização...



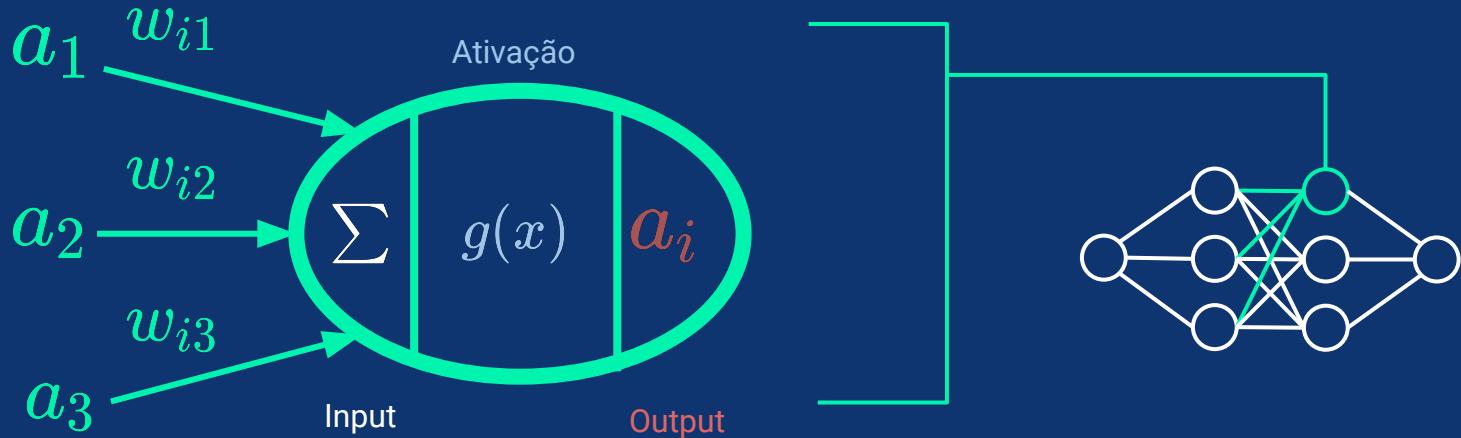
$$\vec{w} \leftarrow \vec{w} - \alpha \nabla C(\vec{w})$$

$$\begin{bmatrix} 0,625 \\ 0,375 \\ 0,2 \end{bmatrix} \leftarrow \begin{bmatrix} 0,5 \\ 0,3 \\ 0,2 \end{bmatrix} + \begin{bmatrix} 0,125 \\ 0,075 \\ 0 \end{bmatrix}$$



BACKPROPAGATION

- Backpropagation é um método para ajustarmos os pesos e bias de nossa rede
- Vamos relembrar como é estruturado um neurônio:



- calculamos a saída com uma “somatória calibrada”:

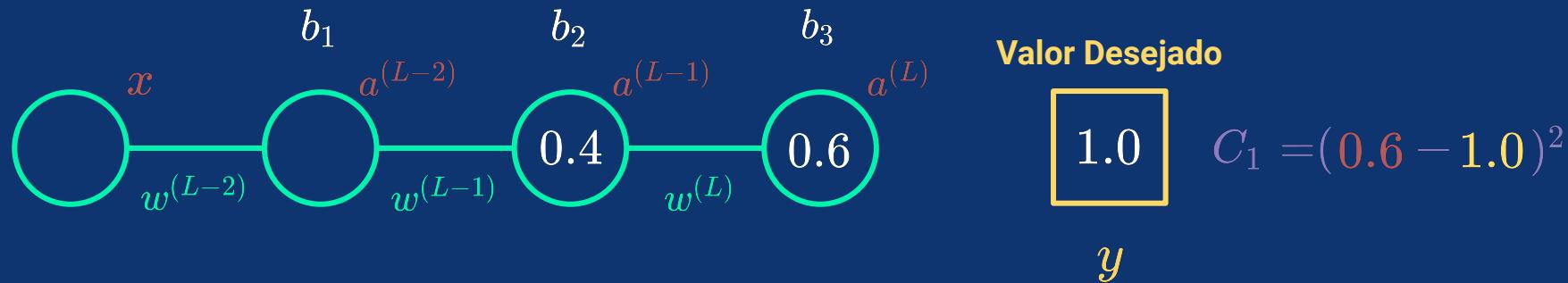
$$a_i = g(a_1 w_{i1} + a_2 w_{i2} + a_3 w_{i3} + b)$$



BACKPROPAGATION

- Lembrando da seção anterior, possuímos uma **função de custo** e queremos minimizá-la.
- Para isso, queremos saber quais ajustes de **pesos** e **bias** melhor minimizam a função de custo. Vamos imaginar um exemplo simples:

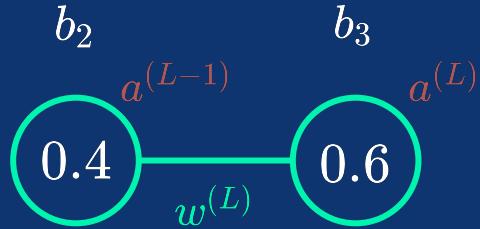
$$\text{Cost} = \frac{1}{n} \sum_{i=1}^n C_i$$



- Vamos calcular então, quanto o peso $w^{(L)}$ é relevante à função de custo



1 - Como chegar do custo até o peso



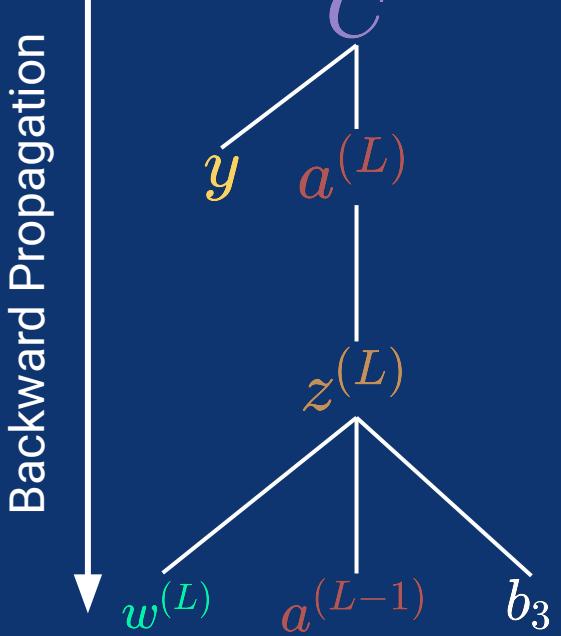
Valor Desejado

 y

$$C_i = (a^{(L)} - y)^2$$

$$a^{(L)} = g(w^{(L)}a^{(L-1)} + b_3) \longrightarrow g(z^{(L)})$$

$$z^{(L)} = w^{(L)}a^{(L-1)} + b_3$$



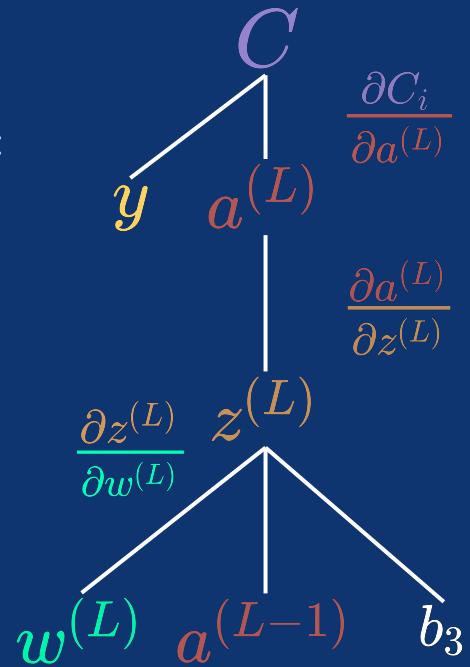
2 - Quanto o **custo** é sensível ao **peso**; a derivada do **custo** em relação ao **peso**

$\frac{\partial C_i}{\partial w^{(L)}}$ → o quanto isso interfere no custo
 $\frac{\partial C_i}{\partial w^{(L)}}$ → uma pequena mudança no peso

- Só que para isso, temos que passar por todas as outras funções também:

$$\frac{\partial C_i}{\partial w^{(L)}} = \frac{\partial C_i}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

- Essa é a **Regra da Cadeia!**



3 - Computando as derivadas

$$C_i = (a^{(L)} - y)^2 \longrightarrow \frac{\partial C_i}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

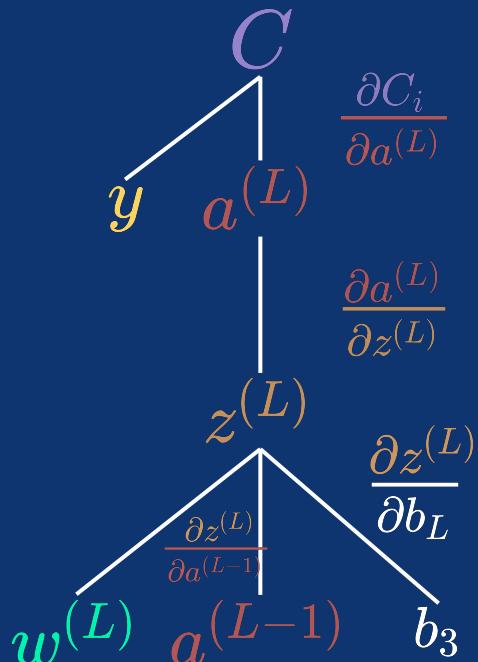
$$a^{(L)} = g(z^{(L)}) \longrightarrow \frac{\partial a^{(L)}}{\partial z^{(L)}} = g'(z^{(L)})$$

$$z^{(L)} = w^{(L)}a^{(L-1)} + b_3 \longrightarrow \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

O quanto o **peso** influencia na
última camada depende do
quão forte é o **neurônio anterior**

$$\frac{\partial C_i}{\partial w^{(L)}} = \frac{\partial C_i}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} = 2(a^{(L)} - y) g'(z^{(L)}) a^{(L-1)}$$

4 - As outras derivadas não são muito diferentes



$$\frac{\partial C_i}{\partial b_L} = \frac{\partial C_i}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial b_L} = 2(a^{(L)} - y) g'(z^{(L)}) \mathbf{1}$$

- Se queremos calcular a derivada relativa ao **bias**, é só trocar a última derivada
- Mesma coisa para a **ativação da camada anterior**

$$\frac{\partial C_i}{\partial a^{(L-1)}} = \frac{\partial C_i}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} = 2(a^{(L)} - y) g'(z^{(L)}) w^{(L)}$$

5 - Para que utilizamos esse resultado?

- Nós apenas calculamos uma derivada de um custo, ainda temos que calcular de todo o treinamento

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_i}{\partial w^{(L)}}$$

- Que é apenas um elemento do vetor gradiente:

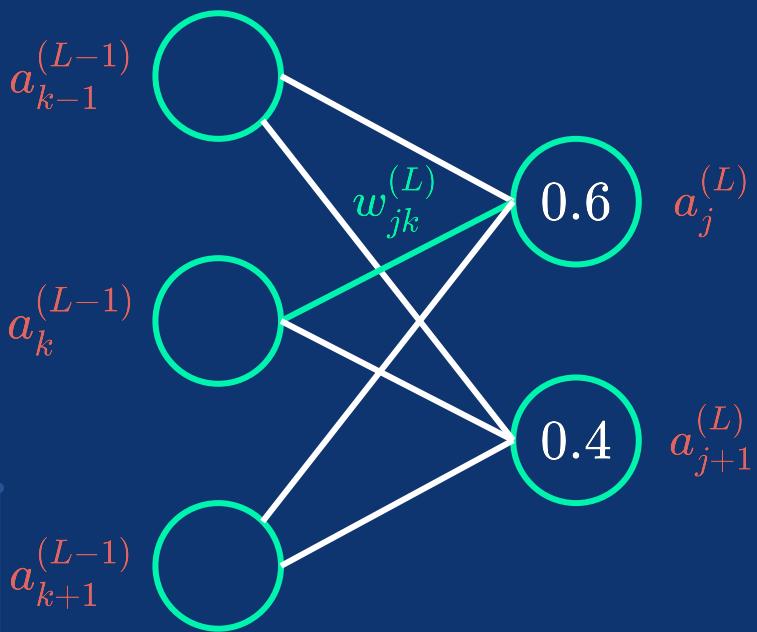
$$\nabla C = \left[\frac{\partial C}{\partial w^{(1)}} \quad \frac{\partial C}{\partial b^{(1)}} \quad \dots \quad \frac{\partial C}{\partial w^{(L)}} \quad \frac{\partial C}{\partial b^{(L)}} \right]$$

- Que repetimos diversas vezes com um **learning rate** para minimizar o erro

$$-\alpha \nabla C$$

GENERALIZANDO

- Podemos ver que muitas das fórmulas são as mesmas
só temos que adicionar alguns índices



$$1.0$$

$$y_j$$

$$0.0$$

$$y_{j+1}$$

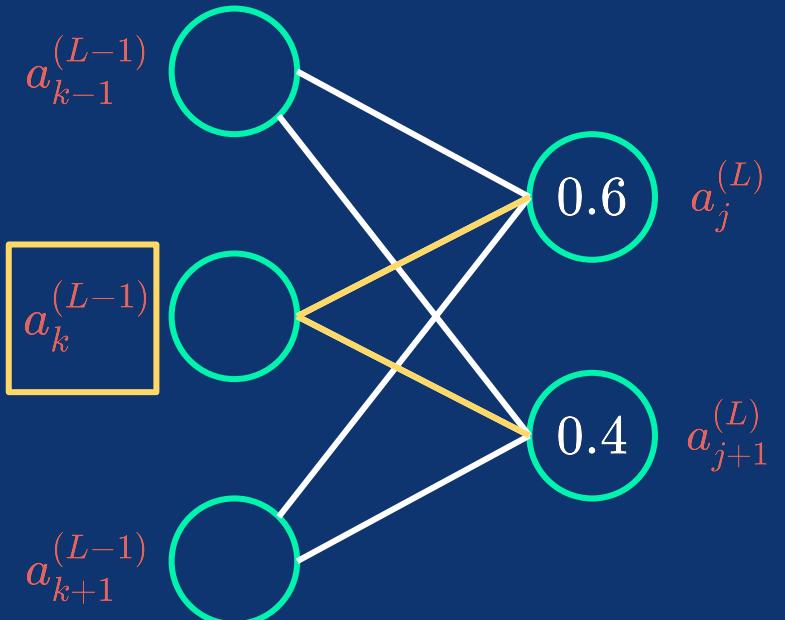
$$C_1 = \sum_{i=1}^n (a_i^{(L)} - y_i)^2$$

$$z_j^{(L)} = b_j + \sum_{i=1}^n w_{ji}^{(L)} a_i^{(L)}$$

$$a_j^{(L)} = g(z_j^{(L)})$$

$$\frac{\partial C_i}{\partial w_{jk}^{(L)}} = \frac{\partial C_i}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$$

GENERALIZANDO



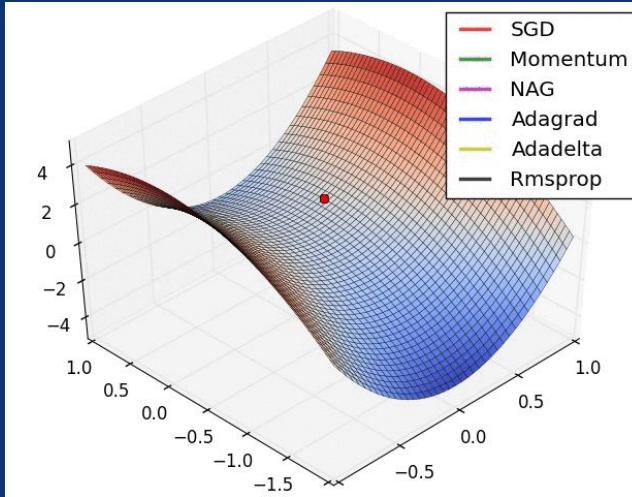
$$\frac{\partial C_i}{\partial a^{(L-1)}} = \frac{\partial C_i}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}}$$

$$\frac{\partial C_i}{\partial a_k^{(L-1)}} = \sum_{j=1}^n \frac{\partial C_i}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}$$

- Temos que realizar essa soma pois a **ativação anterior** influencia as duas **próximas ativações**, que por sua vez influenciam o **custo**

OTIMIZADORES

- Algoritmos de otimização dos pesos de uma rede neural.
- Na prática, vamos utilizá-los já prontos de bibliotecas em vez de programá-los do zero.
- Desempenho de diversos otimizadores:



GRADIENT DESCENT

- O algoritmo de otimização mais simples.

Gradient Descent.

Parâmetros: Taxa de aprendizado $\alpha > 0$, dados de treino S .

Iniciar aleatoriamente pesos \vec{w} da rede neural.

Loop até a rede neural estar suficientemente treinada:

Calcula o custo $C(\vec{w})$ de todos os dados em S :

$$C(\vec{w}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Atualiza os pesos \vec{w} de acordo com a equação:

$$\vec{w} \leftarrow \vec{w} - \alpha \nabla_{\vec{w}} C(\vec{w})$$



STOCHASTIC GRADIENT DESCENT

- Extremamente custoso treinar com todos os dados do conjunto de treino.
- Solução: treinar com somente um de cada vez.

Stochastic Gradient Descent.

Parâmetros: Taxa de aprendizado $\alpha > 0$, dados de treino S .

Iniciar aleatoriamente pesos \vec{w} da rede neural.

Loop até a rede neural estar suficientemente treinada:

Seleciona um dado aleatório de S .

Calcula o custo $C(\vec{w})$ somente deste dado:

$$C(\vec{w}) = (\hat{y} - y)^2$$

Atualiza os pesos \vec{w} de acordo com a equação:

$$\vec{w} \leftarrow \vec{w} - \alpha \nabla_{\vec{w}} C(\vec{w})$$



MINI-BATCH GRADIENT DESCENT

- Treinar somente com um dado gera um alto ruído na otimização.
- Solução: selecionar n dados para serem treinados em mini-batches.

Mini-batch Gradient Descent.

Parâmetros: Taxa de aprendizado $\alpha > 0$, dados de treino S .
Iniciar: aleatoriamente pesos \vec{w} da rede neural.

Loop até a rede neural estar suficientemente treinada:

Seleciona n dados aleatórios de S .

Calcula o custo $C(\vec{w})$ desses n dados:

$$C(\vec{w}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Atualiza os pesos \vec{w} de acordo com a equação:

$$\vec{w} \leftarrow \vec{w} - \alpha \nabla_{\vec{w}} C(\vec{w})$$





DIAGRAMA DE UMA REGRESSÃO LOGÍSTICA

